Project 3-1 Write-up

Yaofu Zuo and Bohan Yu

Link to our webpage:
https://cal-cs184-student.github.io/sp22-project-webpages-BohanYu/proj3-1/index.html

An overview of the project
This project primarily focuses on direct and global illumination through ray-tracing, accelerated by BVH.

Part 1

Ray Generation:

We first compute the lower-left and upper-right positions of the camera. Then, given a normalized location (x, y) in world space, we can convert them into camera space (let's call it x', y', z = -1). The ray we generated originates from camera origin and points toward x', y'. This gives the basic elements (o, d) needed for a ray. However, when we call the constructor for Ray, we must convert (x', y', z) into world coordinates by multiplying the c2w matrix with it. Lastly, we initialize the min_t and max_t of the ray with nclip and fclip.

Primitive intersection:

Given a ray and a primitive "this", we want to see if there's any intersection between the ray and the primitive. We want to find a "t" such that $o + t * d$ will intersect with the primitive. By looking at the computed t, we determine whether the ray intersects with the primitive or not. Specifically, if t < 0, then the primitive is in the opposite direction of the ray, so there can't be any intersections. If t >= 0 yet t is not within the range of min_t and max_t of a ray, we still treat it as no intersection since we can't see it at all, as it's either blocked by other primitives or is simply out of view range.

Triangle intersection algorithm—Möller–Trumbore:

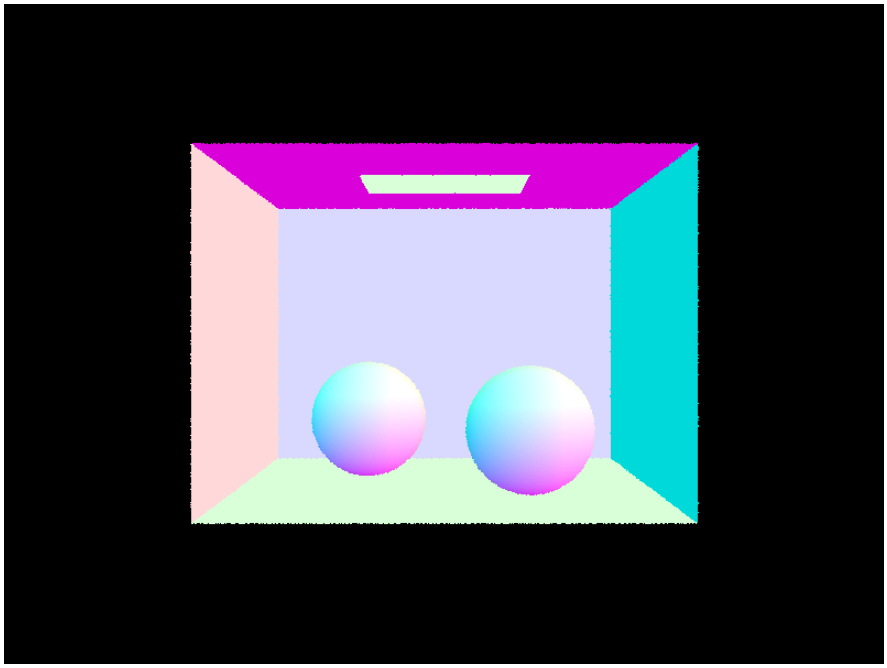
We used the Möller–Trumbore intersection algorithm for determining whether there's an intersection between the ray r and the primitive "this." Though the final equation doesn't make much sense by just looking at it, it's a simplification of the following calculation:

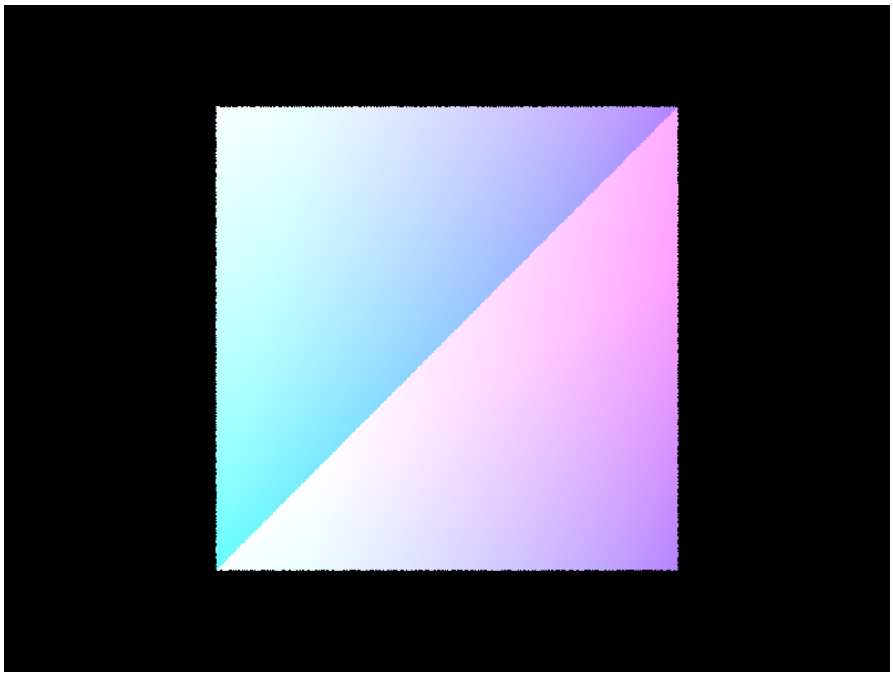
1. Calculate the t at which the ray intersects with the plane. (The plane in which the triangle lies).
2. Calculate the barycentric coordinates of the intersection. (The intersection can be calculated by $o + t * d$).

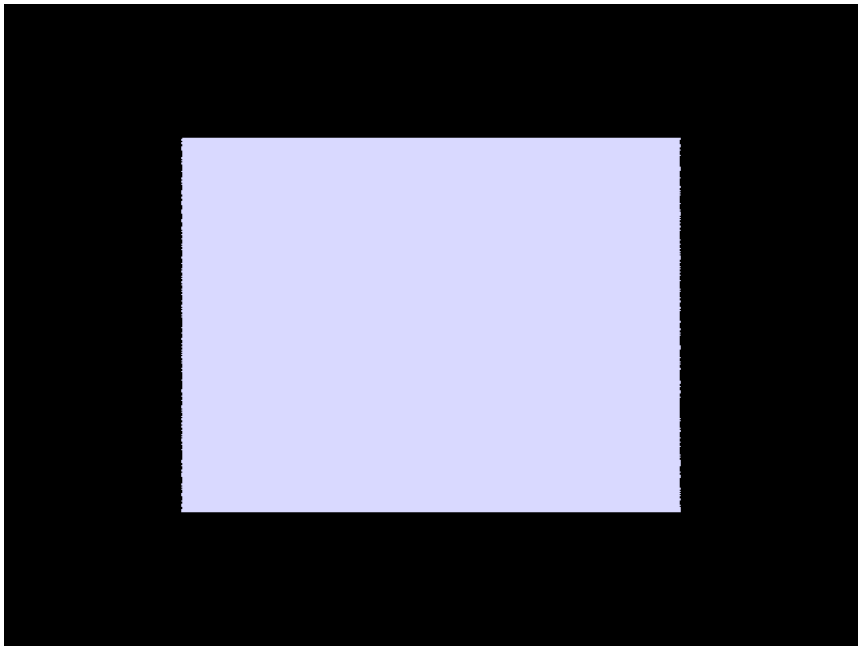
The barycentric coordinates allow us to test whether the intersection lies in the triangle, in which case they must all lie in the range of [0, 1]. We further check that the t is within the min_t and max_t of the ray to make sure that the triangle is within the view range and not blocked by other primitives along the direction of the ray.

In short, we used the Möller–Trumbore intersection algorithm to calculate “t” and the barycentric coordinates of the intersection. We then performed the check above to see if that’s a valid intersection.

Screenshots of some simple .daes: (Spheres, cubes, plane)







Part 2

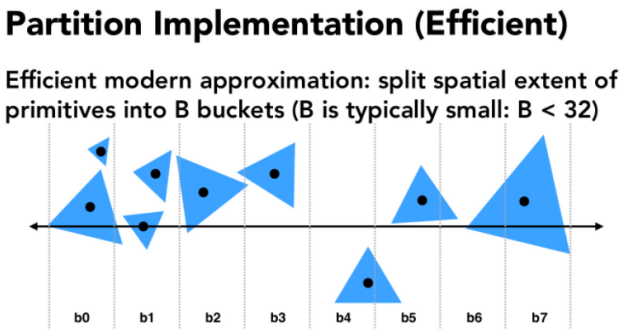
- Walkthrough your BVH construction algorithm. Explain the heuristic you chose for picking the splitting point.
- Show images with normal shading for a few large *.dae* files that you can only render with BVH acceleration.
- Compare rendering times on a few scenes with moderately complex geometries with and without BVH acceleration. Present your results in a one-paragraph analysis.

BVH Construction algorithm:

We first sorted the primitives in increasing order by their x coordinates. Then, for each node, we created its corresponding bounding box by iterating through its primitives using the start and end iterators given. If the number of primitives is not larger than max_leaf_size, then that node is a leaf node and no further action is required. If the number of primitives is larger than max_leaf_size, we split the primitives into “left” and “right” collections, and we chose the median of the centroid of the primitives to be the

split point. We then recursively call the construction function on the left and right collection, therefore recursively constructing the whole BVH.

Extra Credit(Better Heuristic):



We optimized our BVH construction using the partition algorithm above. We first partitioned the node's bounding box into 20 potential split points in the x-axis. For each of those split points, we calculated the cost heuristic using this formula given in lecture :

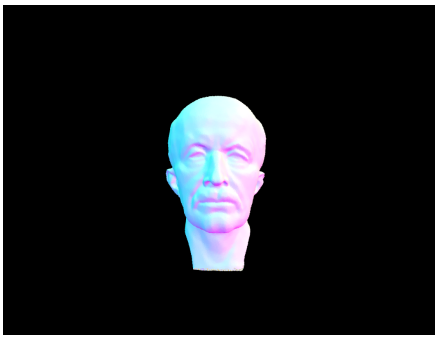
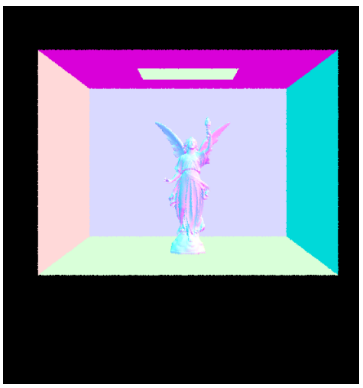
```
Cost(cell) = C_trav + SA(L)*TriCount(L) + SA(R)*TriCount(R)
```

, where the TriCount is the number of Primitives in that cell, and SA is the bounding box of that cell. Within those 20 cost heuristics, we split our BVH based on the split point that gave us the **least** amount of cost.

Performance comparison:

Render Time:

Image	No BVH	BVH with median split point	BVH with Surface Area Heuristic

	82.2399s	1.6809s	1.5537s
	481.3450s	5.7087s	5.3224s
	1548.6934s	1.4370s	1.2690s

The render speed with BVH is significantly faster(at least 50 times faster) than that without BVH. The speed improvement is even larger with symmetrical dae files like the CBlucy.dae, where the render speed with BVH is around 1000 times faster than the render speed without BVH. To further improve the speed, we implemented the surface area heuristic for splitting in BVH. The render time with the SA Heuristic is around 10% less than the render time with median split point.

Extra Credit(Stack instead of Recursion):

BVH Construction:

Instead of recursion, we used a std::stack to store BVHNode pointers. We first initialize the start and end pointers in the root of the BVH tree, and we add the root node to the stack. While the stack is not empty, we popped the top of the stack to get the most recent BVHNode pointer. If the number of primitives is larger than max_leaf_size, we split the node based on the algorithm described in the previous section, and we update the start and end pointers of the left and right nodes and put them into the stack. Else, we do nothing.

Intersection:

Similarly, we replaced recursive calls in intersection with a stack of BVHNodes pointers. We first added the parameter BVHNode *node to the stack. While the stack is not empty, we popped the stack to get our current node, and processed the intersection the same way as recursion.

When using stack for both BVH construction and intersection, the construction time is slightly faster than the recursive implementation by a 0.0003s margin in the cow.dae, using 8 threads 800x600 pixels, where the intersection is much slower than the recursion(from 1.6809s to around 20s in cow.dae). We tried other images and discovered that using the stack in either construction or intersection makes intersection tests significantly slower while the BVH construction did become faster. What's even more confusing is that the **final rendered images were all correct**. We couldn't find the cause for why intersection became significantly slower, because theoretically, recursively calling a function requires many more operations on the stack (x86 stack) than iterations. We do suspect that using iterators when constructing BVH with stack could make referencing significantly slower, but we are not sure about this conclusion.

To prove that we did try to implement stack, here is the github branch where we implemented the stack version of both BVH construction and intersection:

https://github.com/cal-cs184-student/p3-1-pathtracer-sp22-ai-loves-ml/tree/2-1-op

Part 3

Direct Lighting with Uniform Hemisphere Sampling:

We used the Monte Carlo estimator with uniform hemisphere sampling to approximate the reflection equation.

We uniformly sample "num_samples" times over a hemisphere. For each iteration (for each sampling), we first sample an incoming direction by calling `this->hemisphereSampler->get_sample();` and convert the returned result into world space. This allows us to construct a sample ray based on the hit point and this direction. We want to see if this ray intersects with any primitives. If it intersects, we compute $f(w_{out}, w_{in}) * Light\ emitted\ at\ the\ intersection * cos(\theta_{w_{in}})$ and add it to L_{out}. After all iterations end, we return $2 * \pi * L_{out} / number\ of\ samples$. The returned value is the Monte Carlo estimator for the reflection equation, with uniform hemisphere sampling.

Direct Lighting by Importance Sampling Lights:

Instead of sampling over a hemisphere, this method directly samples light sources. This allows us to accommodate point lights, in which case the uniform hemisphere sampling will simply return 0.

For each light source in the scene, we sample it "num_sample" times. "Num_sample" depends on whether the current light source is a point light or not. If it's a point light, then we only need to sample once (since each sample will be the same). Otherwise, we sample `ns_area_light` times, which stores the number of samples per area light source.

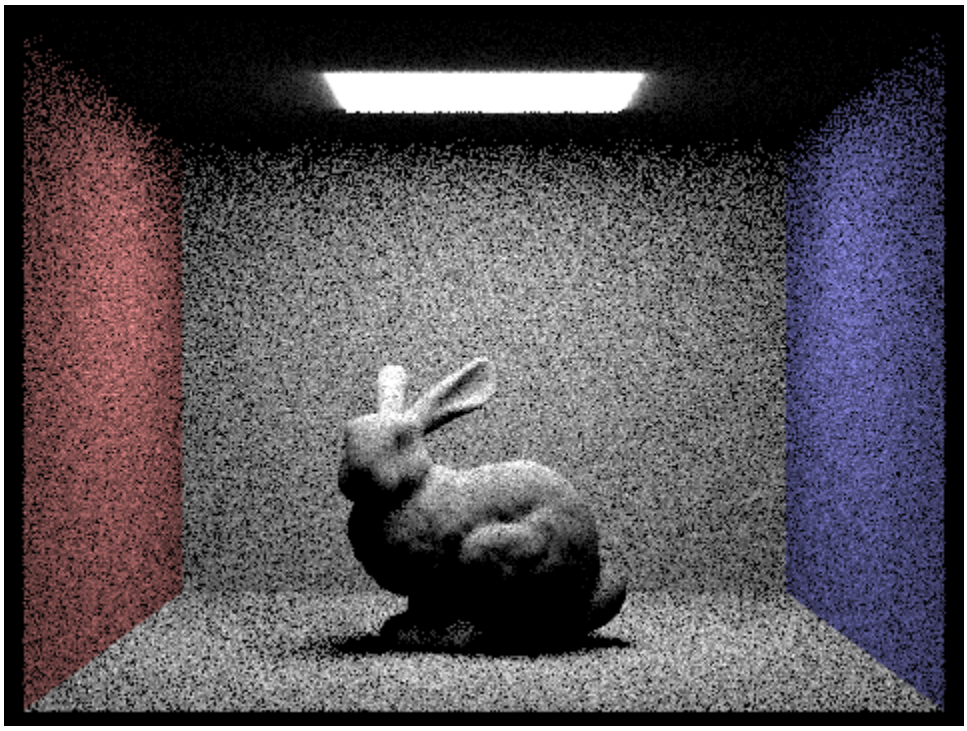
Inside each sampling iteration, we first sample the light by calling its instance function `sample_L` ,which tells us the emitted radiance at the hit point, with a sampled direction of w_{in} in world coordinates. It also tells the probability of this sample, which will be used when we update L_{out}. If the light is not blocked by the object from the camera view, we construct a ray that originates from the hit point and points toward w_{in}, and set its max_t to be the distance between the light source and the hit point (this is returned by

sample_L). An implementation detail here is that we need to set the min_t of the ray to be a very small non-zero constant: EPS_F. We also subtracted EPS_F from the max_t. We test whether this ray intersects with any other primitives. If not, then we know that the light can safely arrive at the hit point without being blocked and we add L_{out} with $Light\ emitted\ by\ the\ intersection\ *\ f(w_{out}, w_{in})\ *\ cos(\theta_{w_{in}})\ /\ probability\ of\ this\ sample$. After the sampling process ends for each light source, we need to weight L_{out} with $L_{out} = L_{out}\ /\ number\ of\ samples$. This completes the whole sampling iteration for one light source. After we repeat the same process with all the sampled light sources, we return L_{out}.

Comparisons between the two implementations:

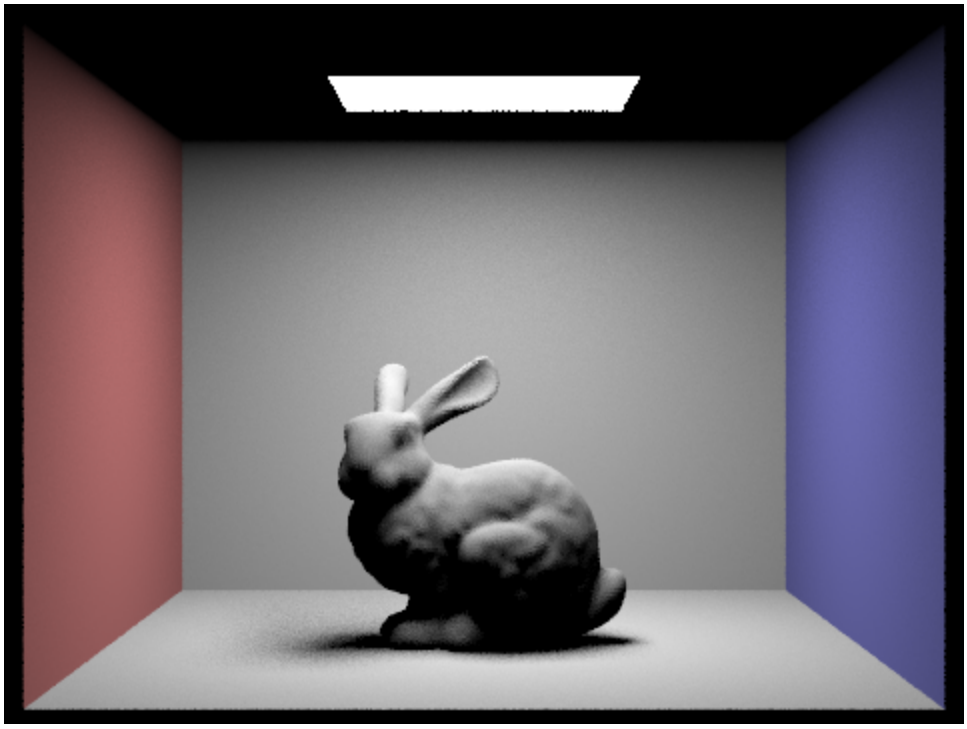
A Bunny rendered with Uniform Hemisphere Sampling:

-t 8 -s 16 -l 8 -H -f CBbunny_H_16_8.png -r 480 360 ../../dae/sky/CBbunny.dae



The exact same bunny as above except now using Importance sampling:

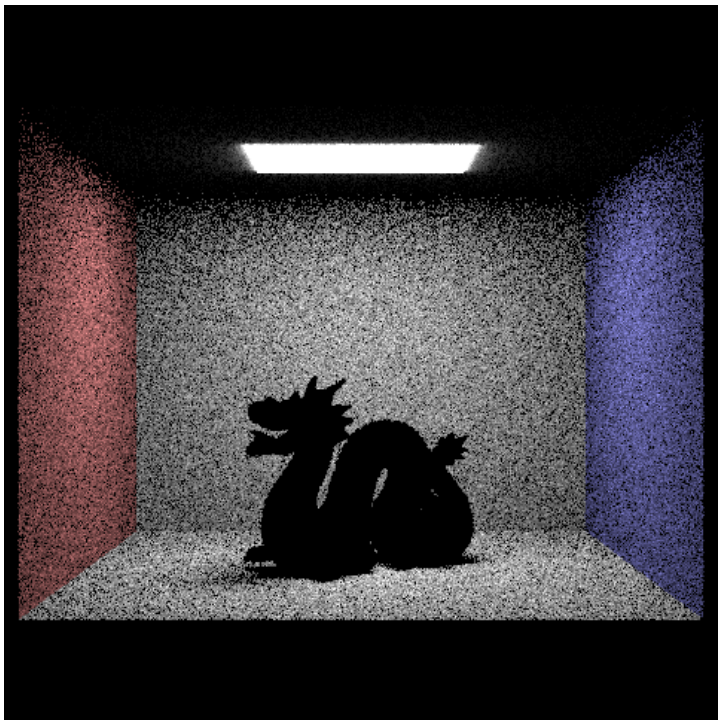
-t 8 -s 16 -l 8 -f CBbunny_Importance_16_8.png -r 480 360 ../../dae/sky/CBbunny.dae



A Dragon rendered with Uniform Hemisphere Sampling:

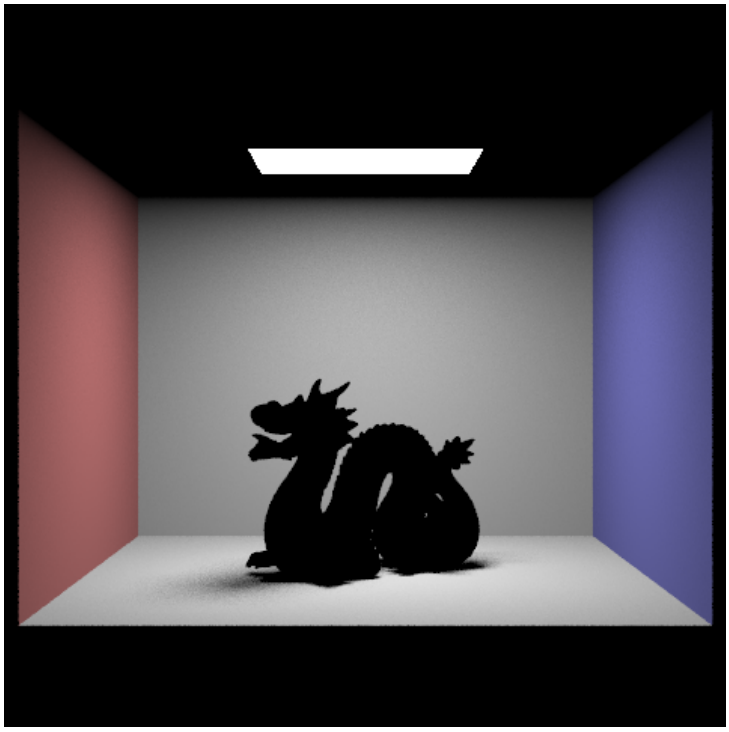
-t 8 -s 16 -l 8 -H -f dragon.png -r 480 480 ../../dae/sky/CBdragon.dae

(Both “s” and “l” are the same as the bunny. The spec used s = 64. My computer is too slow to render that unfortunately TT.)



The exact same Dragon as above except now using Importance sampling:

-t 8 -s 16 -l 8 -f dragon_importance.png -r 480 480 ../../dae/sky/CBdragon.dae

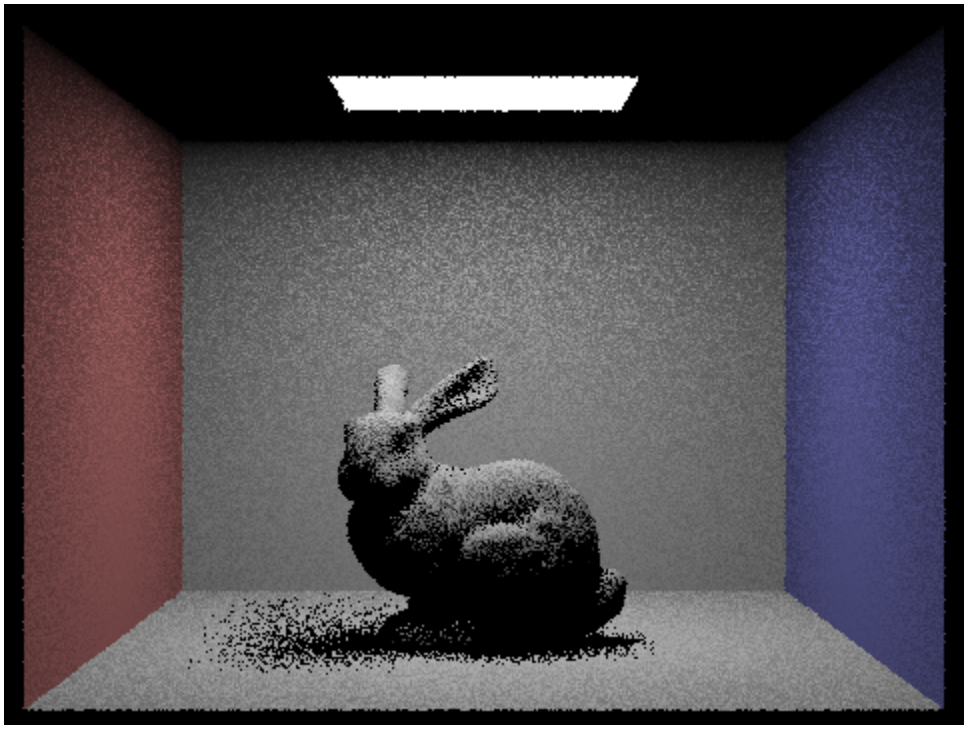


Analysis:

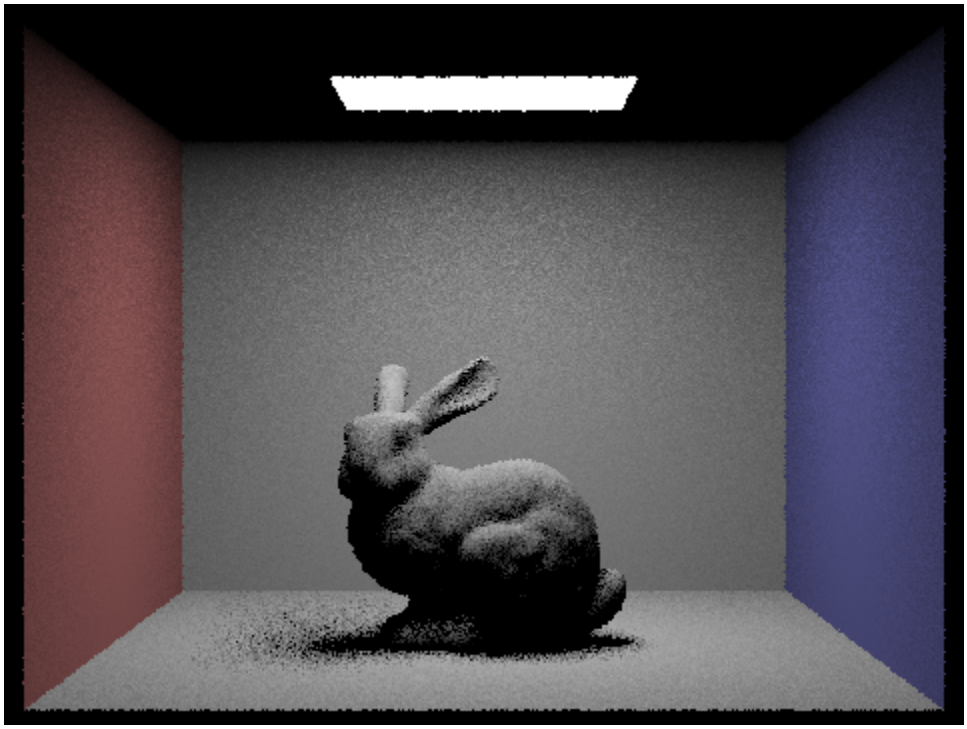
With importance-light sampling, the whole scene looks more smooth and clean. But when it comes to uniform hemisphere sampling, the whole scene looks more noisy, dotted by a lot of black dots. Such noise is expected, as the uniform hemisphere sampling samples random directions on the hemisphere, and it's likely that for one pixel, most of its randomly sampled directions hit the light source whereas a neighboring pixel is completely black (i.e. none of the sampled direction hits the light source). This makes pixels whose hemisphere is not completely blocked from the light source to become noisy, resulting in the noise we see. In contrast, the importance-light sampling only samples over the area of the light source (i.e. only samples over directions that could hit the light source) instead of the whole hemisphere. This eliminates the noise since for pixels where only certain directions could hit the light source, we can no longer sample directions where no light is present (possibly resulting in a completely black dots even if some light source could arrive at that pixel), but instead we can only sample from the directions that could hit the light source, guaranteeing that the pixel will receive the amount of light proportional to the amount of directions toward the light source that are not blocked.

Comparisons between different light sample rate using Light Sampling:

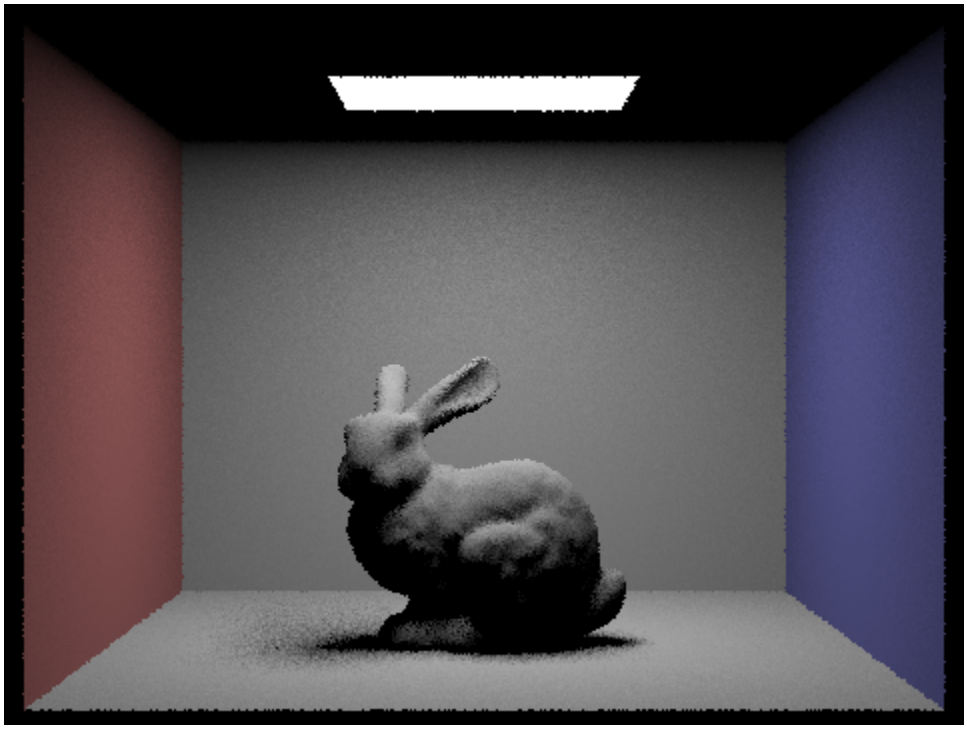
CB Bunny rendered with L = 1:



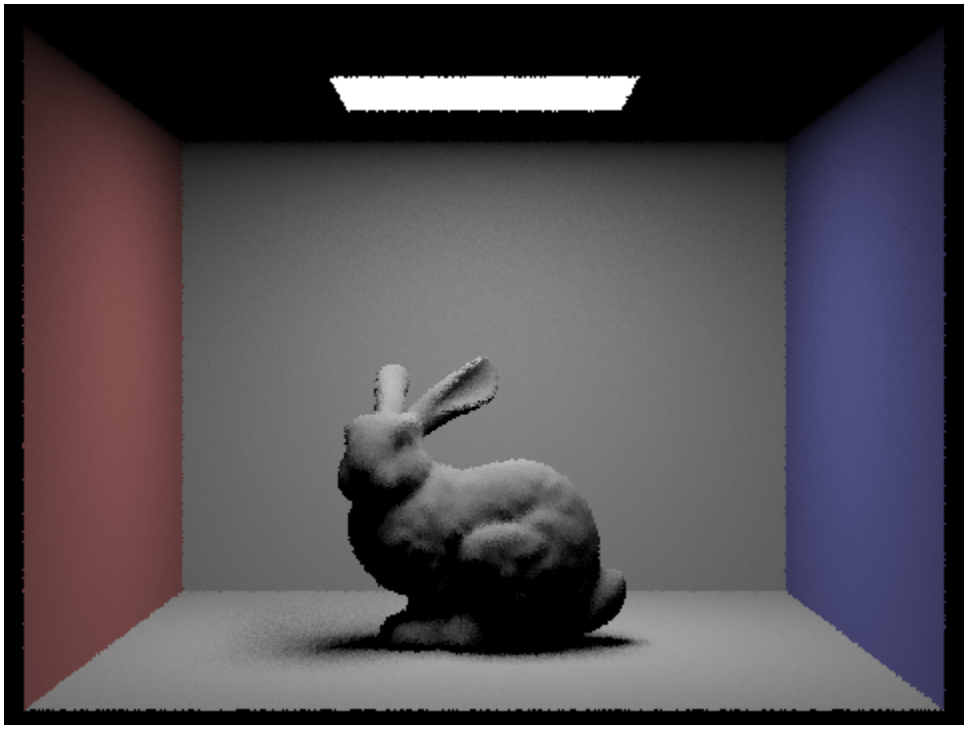
CB Bunny rendered with L = 4:



CB Bunny rendered with L = 16:



CB Bunny rendered with L = 64:



The noise level in the shadow area decreases as we increase the light sampling rate. (i.e. reduced variance in the final light computed)

As shown in Picture 1 with L = 1, there are lots of black dots spreaded over a large area, whereas in the last picture with L = 64, the area of the totally black shadow looks significantly smaller. This is because when we calculate the light received at a pixel, it's no longer "all-or-none" as in the case of L = 1. We cast more rays and average the light across all samples, making it more likely for a pixel to receive some lights and reducing the variance. So the shadow far away from the Bunny is no longer dotted by completely black dots or completely gray dots (i.e. the color of the surroundings), but instead populated by a "less black" color, where some rays are blocked and others are not. This makes the shadow appear to be less noisy, as the shadow area now blends more smoothly into the surroundings.

Part 4

Indirect Lighting Implementation:

At a high level, in order to simulate global illumination, we need to add together the one-bounce radiance and radiance that takes multiple bounces, where the maximum number of bounces are determined by the parameter "m." At the beginning of the function, we check if the ray's depth is smaller than 1, in which case we have reached the base case and simply return 0. Otherwise, we initialize `L_out = one_bounce_radiance(r, isect)`. For safety, we also check if the max_ray_depth is smaller than equal to one. If so, we return L_out that is equal to the one_bounce_radiance. Otherwise, we continue and see if the Russian Roulette tells us to terminate. If the Roulette tells us to terminate and the ray's depth isn't equal to max_ray_depth (i.e. if m > 1, then we don't want to end with one_bounce_radiance), we return L_out that is equal to the one bounce radiance. Otherwise, we construct a new ray that originates from the hitpoint with a sampled incoming radiance direction by calling sample_f. We also initialize this new ray's depth to be the original ray's depth minus 1 such that the recursion will end when we hit the base case. A minor detail here is that when we initialize a camera ray, we set its depth to be the max_ray_depth so that we will at most trace "max_ray_depth" times with our scheme. We want to see if this newly constructed ray will intersect with any primitives or not. If it intersects, then we recursively call the function at_least_one_bounce_radiance with this new ray and the new intersection, and we

update L_out accordingly. If the newly constructed ray doesn't intersect anything, then we do nothing. Lastly, at the end of the function, we return L_out. (We attached our Pseudo code below)

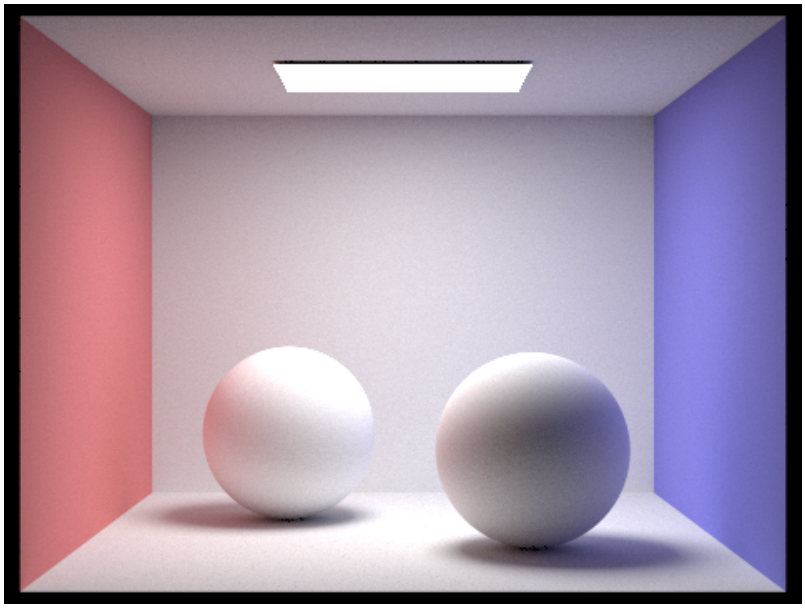
Here is our pseudo code:

```
// Base case.
if (r.depth < 1) {
   return a vector of 0;
}

// One bounce case.
L_out = one_bounce_radiance(r, isect);
if (max_ray_depth <= 1) {
   return L_out;
}

// At least one bounce case.
// Termination probablity.
double p = 0.3;
double prr = 1 - p;
bool terminate = coin_flip(p);
Sample a new outgoing direction at the hitpoint. (Even though it's called w_in)

if (!terminate || r.depth == max_ray_depth) {
        Construct a new ray.
   if (the new ray intersects with any primitives) {
      Recurse and Update L_out;
   }
}
return L_out;
```

Some images rendered with Global Illumination:

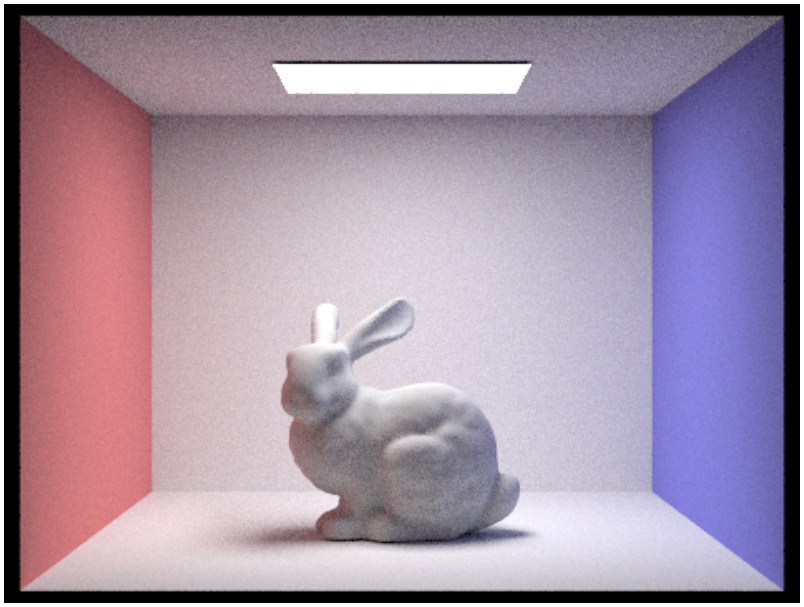
CBspheres:

-t 8 -s 1024 -l 16 -m 5 -r 480 360 -f Part4_spheres.png
../../dae/sky/CBspheres_lambertian.dae



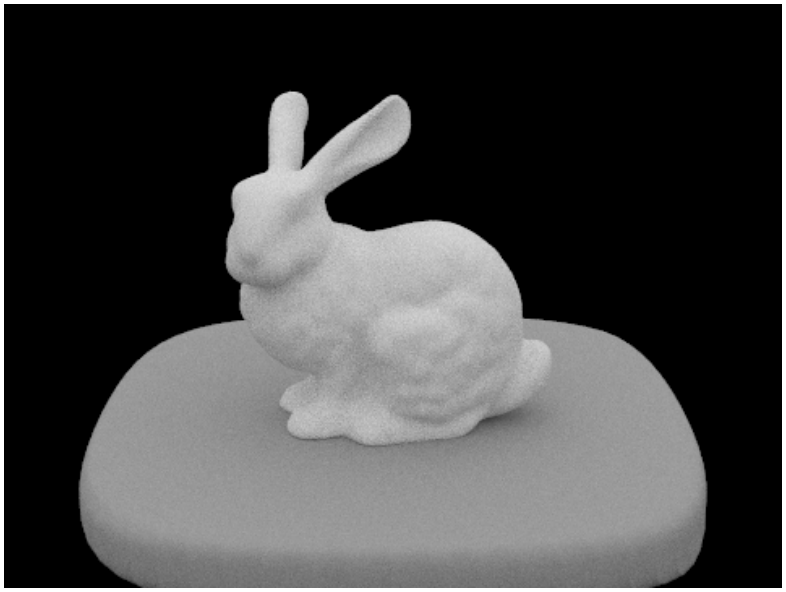
CB Bunny:

-t 8 -s 1024 -l 4 -m 5 -r 480 360 -f Part4_Bunny.png ../../dae/sky/CBbunny.dae



Bunny: (not CBbunny, just Bunny)

-t 8 -s 64 -l 4 -m 5 -r 480 360 -f Part4_bunny.png ../../dae/sky/bunny.dae

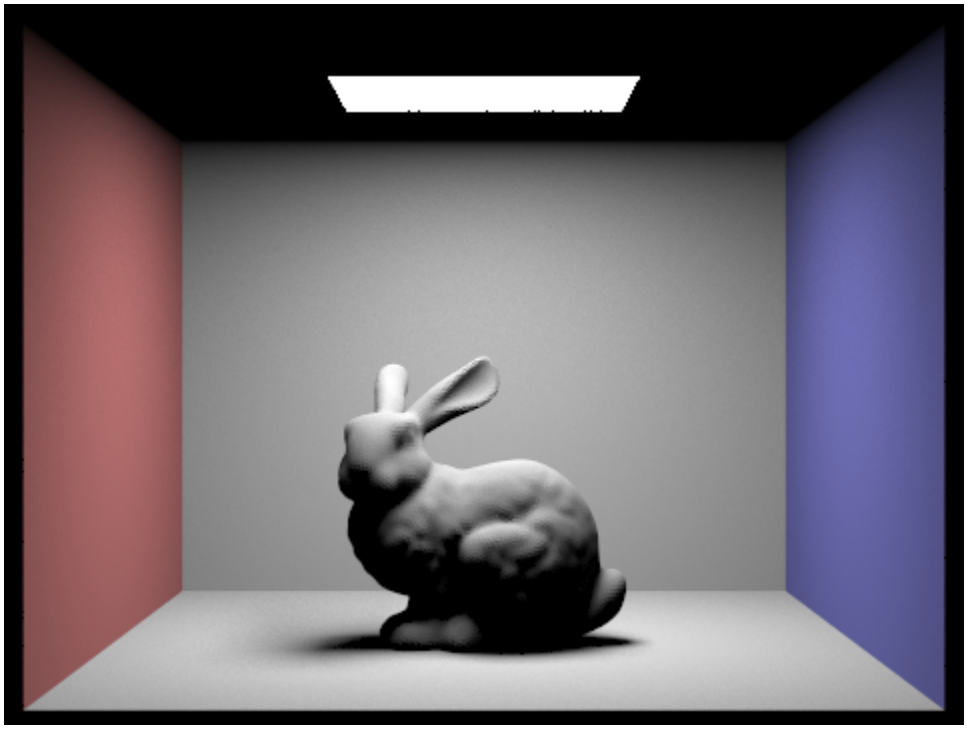


CBbunny, rendered with different max_ray_depth:

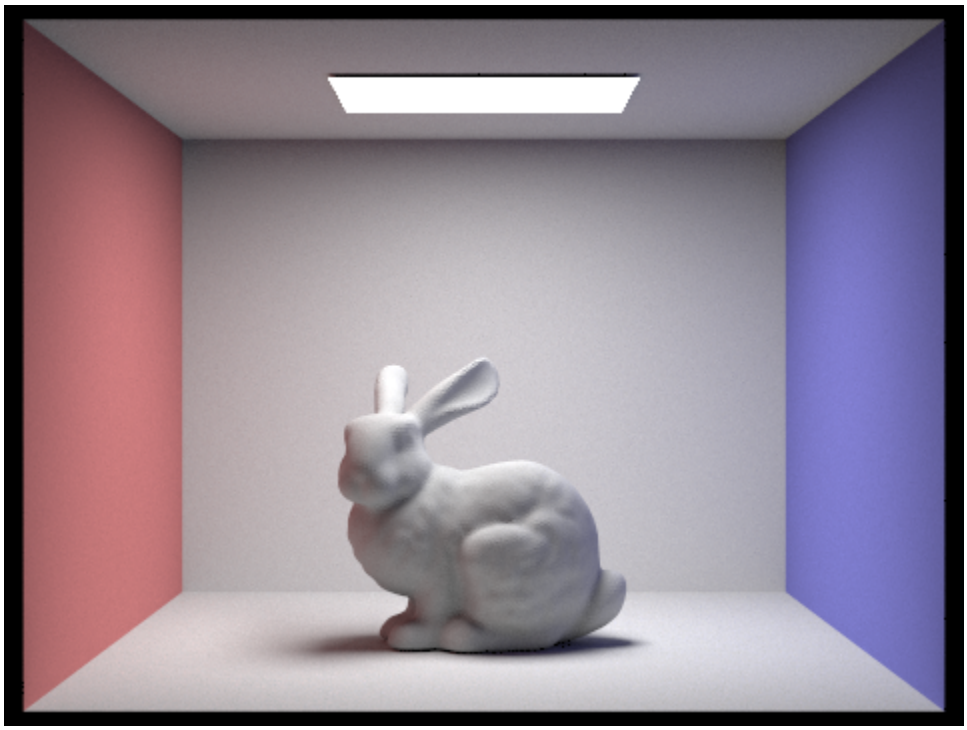
(Note: we used a L = 4, that is, light sample rate = 4.)

Max_ray_depth = 0:

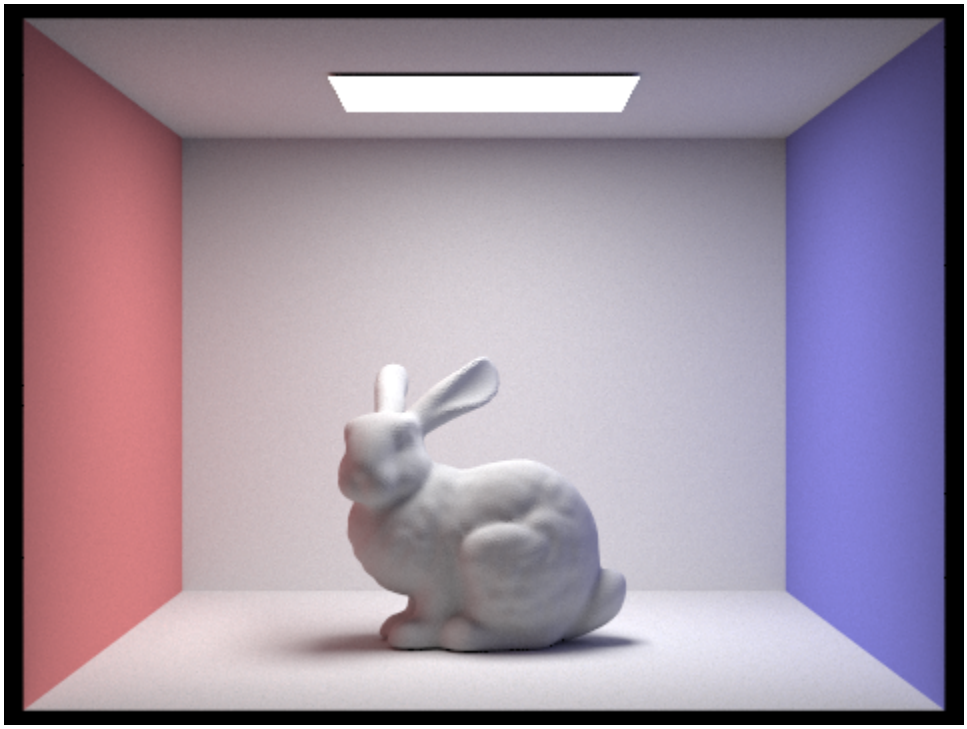
Max_ray_depth = 1:



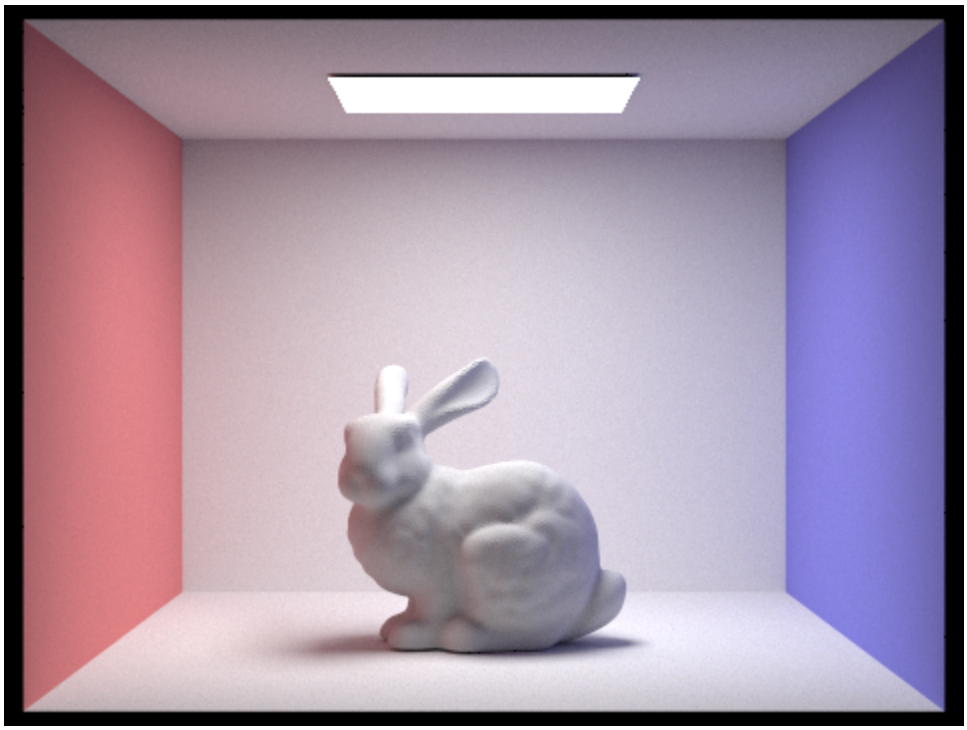
Max_ray_depth = 2:



Max_ray_depth = 3:



Max_ray_depth = 100:

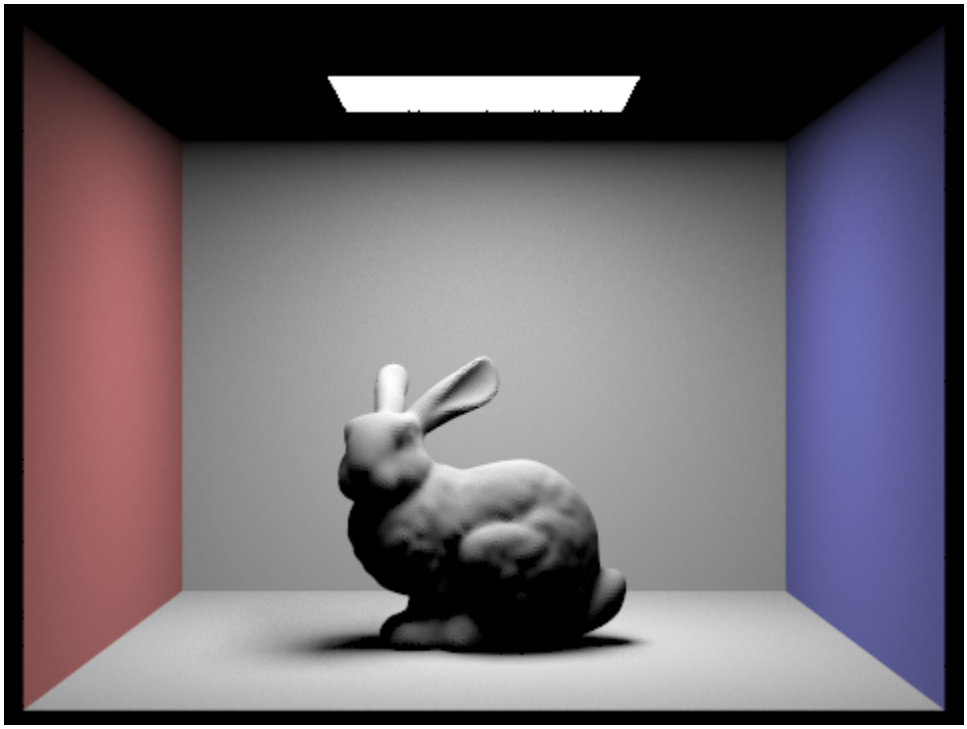


Comparison:

As we increase the max_ray_depth, the whole scene becomes brighter and brighter, and should theoretically become more and more realistic. When M = 0, it's simply zero-bounce illumination. Similarly, when M = 1, it's just one-bounce illumination. As M goes beyond 1, the whole scene starts incorporating lights that need multiple bounces to arrive at the camera, thus making the scene more realistic. It might be hard to notice the difference between M = 100 and M = 3, because the more the light bounces, the less influence it has on the final rendered result. We also have a Russian Roulette "terminator" that makes it hardly possible for a light to really bounce 100 times. However, even though the difference is small, we can still notice some areas in M = 100 that are slightly brighter than those rendered with M = 3. For example, we can notice that when M = 100, more of the pinkness on the left wall and the blueness on the right wall gets reflected onto neighboring walls, especially at corners. So the corners at M = 100 are more colorful and thus brighter than those rendered with M = 3.

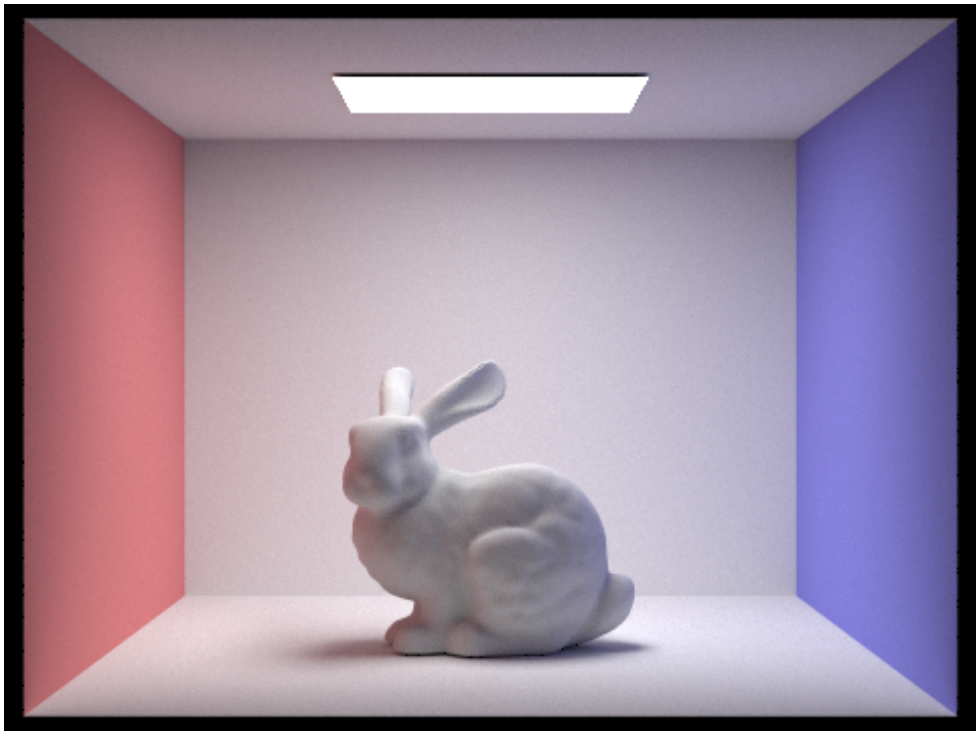
Different illumination:

CB Bunny rendered with direct illumination only: L = 4, S = 1024.



CB Bunny rendered with indirect illumination only:

M = 5, L = 4, S = 1024.

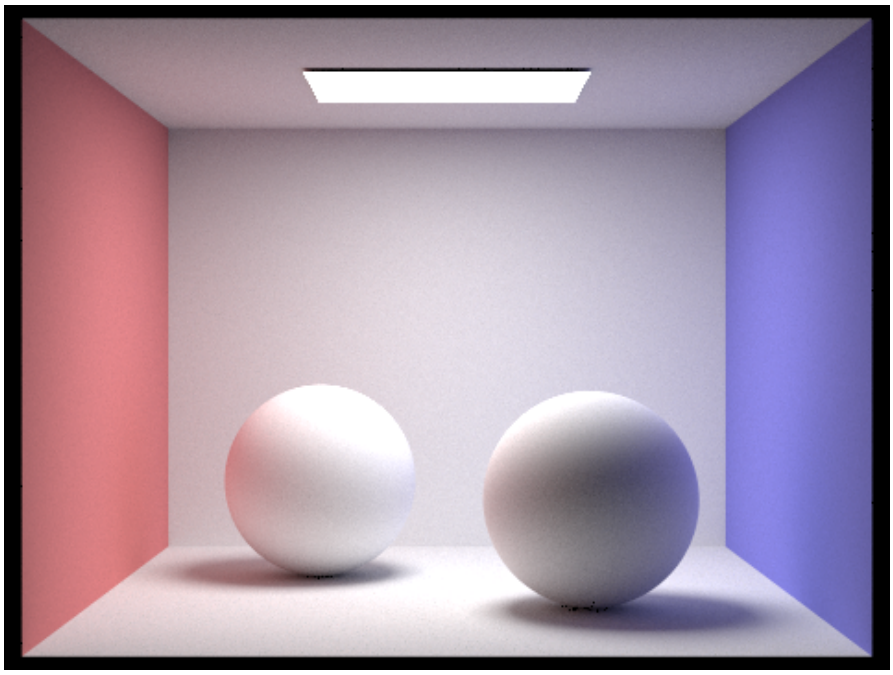


Comparison:

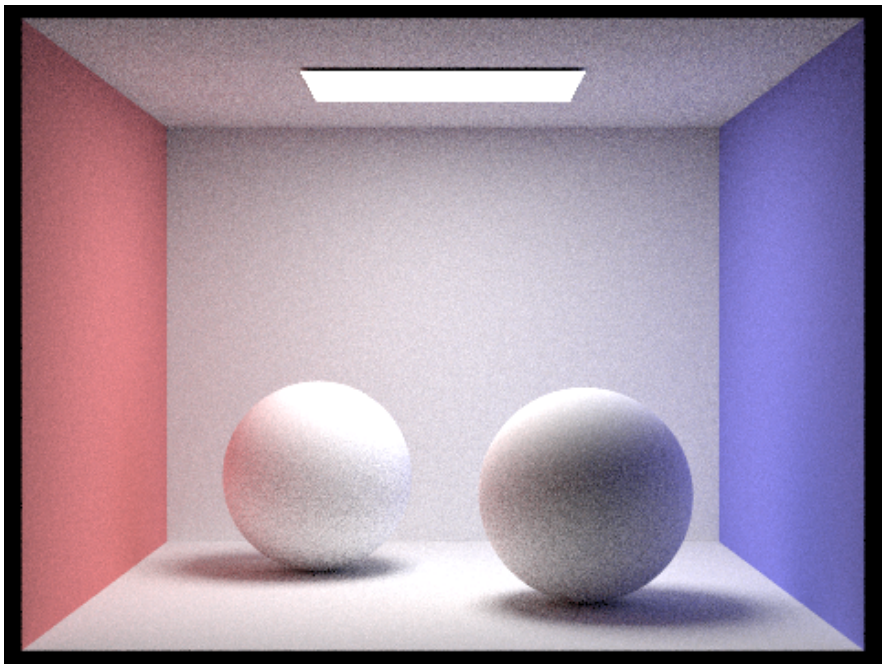
With direct illumination, most parts of the bunny and the whole surroundings appear to be darker than the one rendered with indirect illumination only, since direct illumination can only give very limited lighting to the surroundings and the bunny. The bunny rendered with indirect illumination only may appear to be the same to the one rendered with both direct and indirect illumination. However, if we zoom in at the bunny's ear and compare it with the one rendered with global illumination, we can still notice that the indirect illumination renders a slightly darker image. (We used OpenCV to read in the rendered png and compared their brightness.) Nevertheless, this result shows that most of the lighting in this scene actually comes from indirect illumination, instead of direct illumination.

CB Spheres rendered at L = 4, with different sample rate:

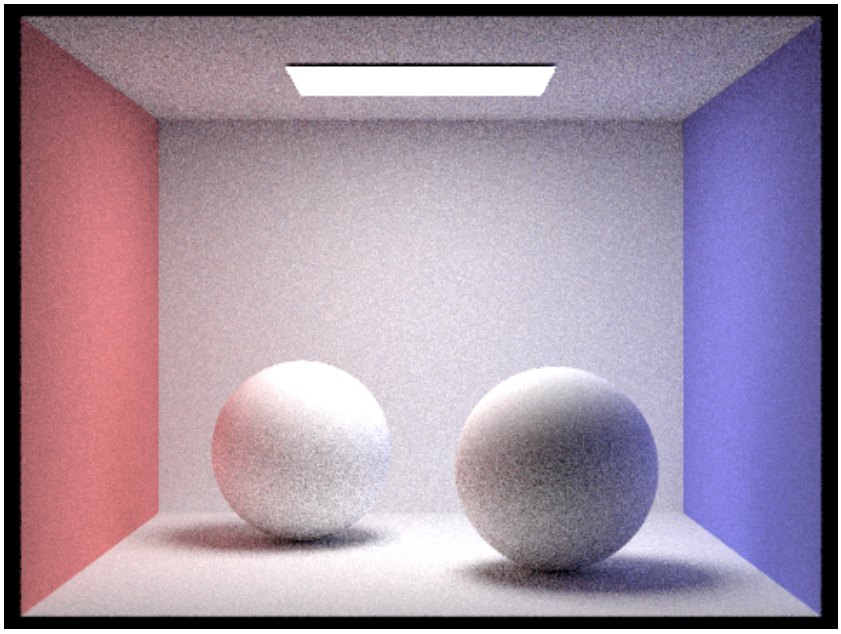
S = 1024:



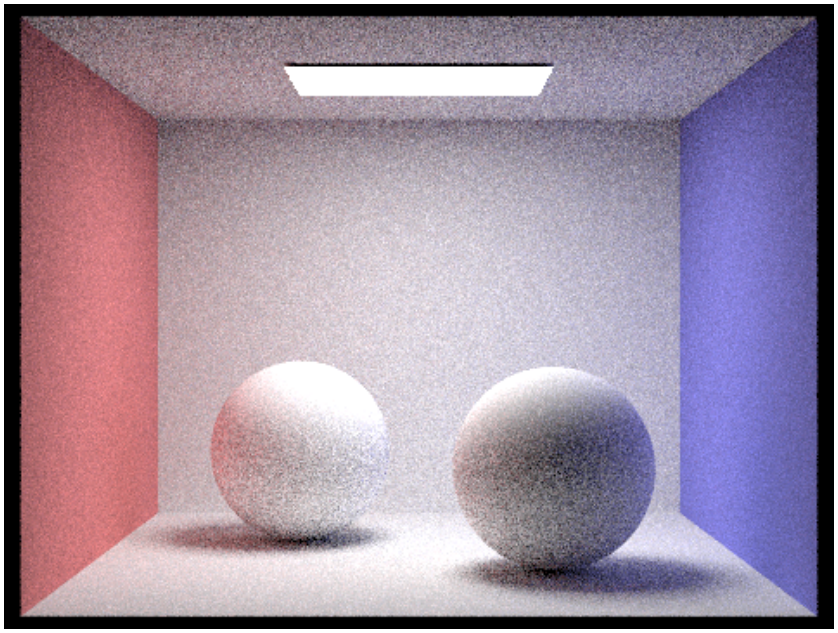
S = 64:



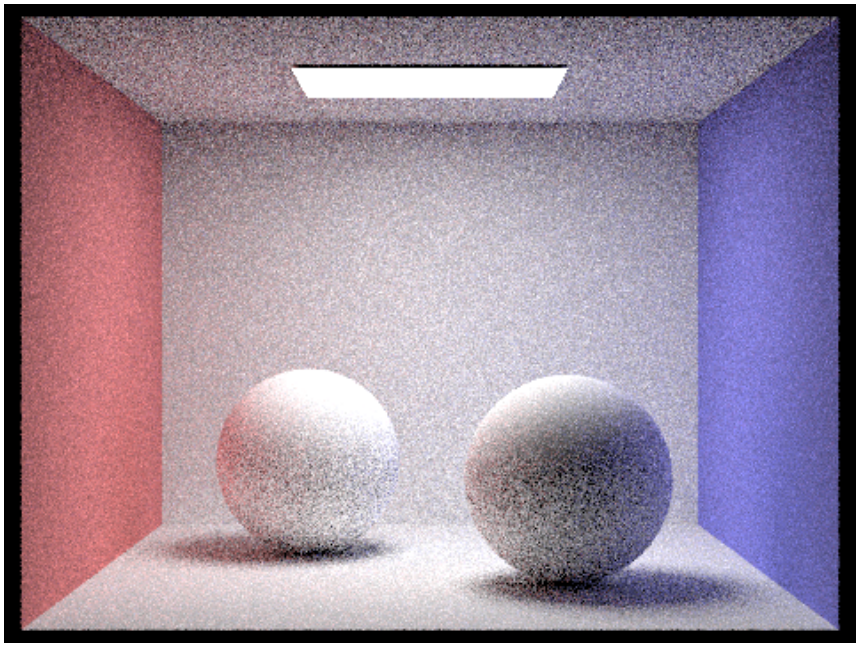
S = 32:



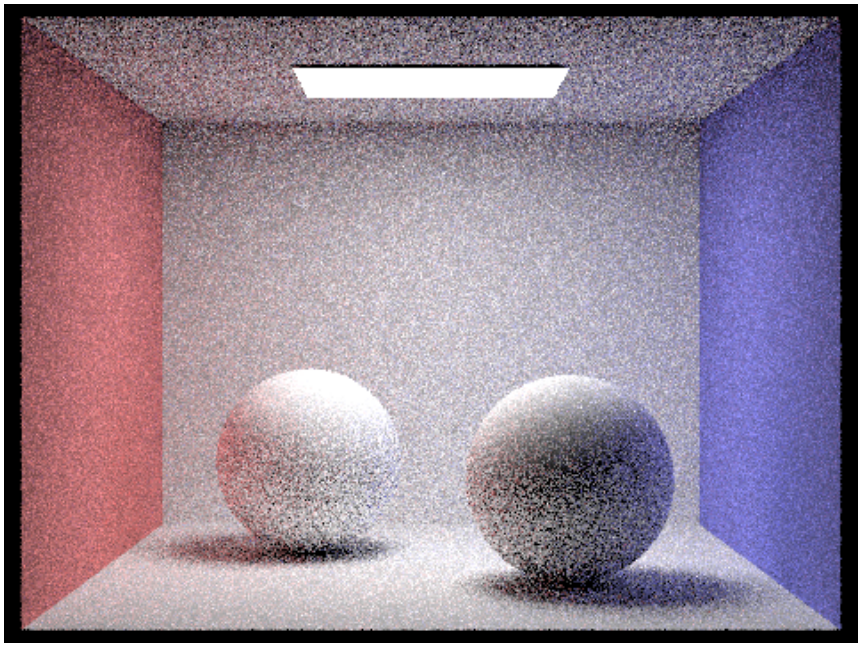
S = 16:



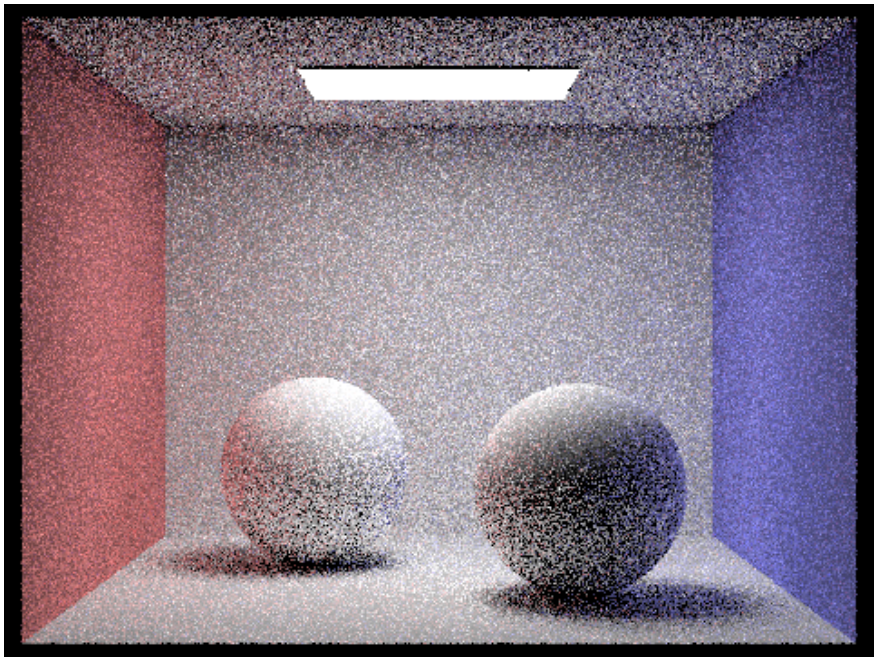
S = 8:



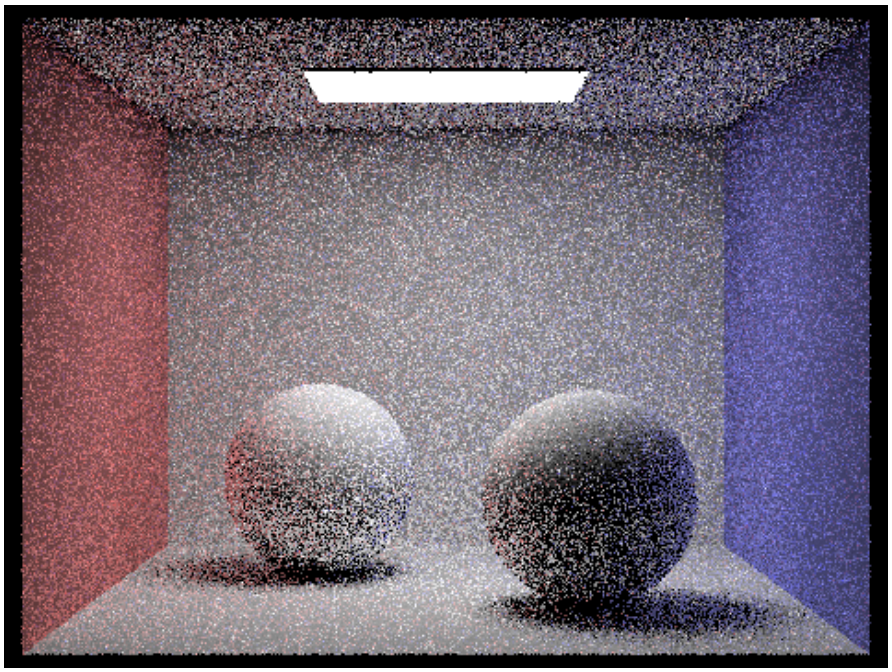
S = 4:



S = 2:



S = 1:



Comparison:

As we decrease the number of camera rays per pixel, we can clearly see more noise in the rendered image due to the randomness in our sample of carema ray direction. Since

we sample different random carema ray directions for each pixel, only some of those directions are pointing towards the light source(so only some tracing rays would hit the light source while others won't). Thus, as the number of carema rays decreases, the lower the probability that the sample you select is going to hit the light source that shines light at that pixel, and the lower the probability that this pixel's brightness and color would be calculated correctly. As a result, the accuracy of the image would decrease when we decrease the number of carema rays, creating more random noise and darker images like the images above.

Part 5

For every iteration of sampling, we calculated x_k, the current sample's illumination, and updated our variables s1 and s2 according to the formulae provided by the lecture.

$$s_1 = \sum_{k=1}^{n} x_k, \qquad s_2 = \sum_{k=1}^{n} x_k^2,$$

Then, at every samplePerBatch(64) iterations, we calculated μ and σ according to the following formulae:

$$\mu = \frac{s_1}{n}$$

$$\sigma^2 = \frac{1}{n-1} \cdot \left(s_2 - \frac{s_1^2}{n} \right)$$

At last, we calculated variable I, which is equal to "1.96 * σ / √(n)", and we checked if I is less than or equal to maxTolerance * μ. If so, we break out of the sampling for loop and update our pixel. If not, we continue sampling another 64(or samplePerBatch) times.

CBunny.dae rendered at 2048 carema rays per pixel, 64 samples per batch, 0.05 tolerance for adaptive sampling, 1 sample per area light, 5 maximum ray depth, and resolution of 480x360: